C/C++ Contributing Editors

The New C: VLAs, Part 4: VLA typedefs and Flexible Array Members

Randy Meyers

The Rest of the Story on variable-length arrays in C99. Yes, they're well-behaved and very flexible, but use them with caution.

My last few columns have dealt with VLAs (Variable Length Arrays) in C99 [1, 2, 3]. VLAs are arrays with run-time expressions instead of compile-time constant expressions for the bounds of the array. The bounds expression is evaluated when the declaration of a VLA is reached inside of a block, and the array has the calculated bounds until its lifetime ends (usually by exiting the block).

This column discusses the remaining feature of VLAs, VLA **typedef**s. I will also discuss flexible array members, a C99 feature similar to VLAs.

VLA typedefs

As I discussed in previous columns, the size of a VLA is needed at run time to perform indexing and address arithmetic, so the compiler must make arrangements to store the size of the array somewhere. However, the size is not stored in the array object itself. It is not stored as part of the pointer if you have a pointer to a VLA. The size of a VLA is an attribute of the VLA type [3].

Consider the following:

```
void ex1(int n)
{
    char (*pvla)[n];
     n += 10;
    printf("%zu", sizeof *pvla);
}
```

pvla is a pointer to a VLA of **n** **char**s. In order to do pointer arithmetic with **pvla** or in order to be able to return the size of the objects to which **pvla** points, the compiler must calculate the size of a VLA of **n** elements of type **char**. Since the C99 rules say that a VLA's size is fixed at the point the declaration of its type is encountered, the compiler must perform the calculation of the array's size at the point of the declaration to protect against the value of the bounds expression changing later in the program. The function **ex1** prints the size of the array to which **pvla** points. Since the size of an array of **n** elements of type **char** is just **n**, that is the value that the function prints. However, it prints the original value of **n** passed to the function, not the value of **n** after **10** has been added to it.

Note that the function **ex1** works even though **pvla** is uninitialized stack trash. The program is perfectly valid because **sizeof** does not actually evaluate its argument: the

uninitialized pointer **pvla** is not actually dereferenced. The **sizeof** operator only inspects its operand in order to determine the resulting type, and in C, size is an attribute of the type of an expression. The function **ex1** makes this clear. **pvla** does not actually point at an array, so the size information could not be stored as part of the array object. Likewise, **pvla** is uninitialized stack trash, so the size information could not be part of its value.

Instead, compilers generate code to record the size of VLA types in the program, not the VLA objects themselves. For every VLA type that occurs in a block, the compiler creates an unnamed automatic temporary variable that holds the size of that VLA type during its lifetime. When the type is executed by program flow of control reaching a declaration or cast involving a VLA type, the size of the VLA type is stored in the temporary variable. If the size of a VLA is needed, then the value is fetched from the temporary variable associated with the VLA type. When the block containing the VLA type exits, then the temporary variable is deallocated along with all of the other automatic variables.

Of course, a clever compiler might not create a temporary for every VLA type in a block. If the compiler can prove that several of the temporaries always hold the same value or that the temporaries are not used later in the block, the compiler might optimize them away.

Clearly, C99 compilers are proficient in handling the bookkeeping associated with VLA types. The C99 language builds upon that by allowing VLA **typedef**s.

```
void ex2(int n)
{
    typedef int VARRAY[n];
    n += 10;
    VARRAY a1, a2;
}
```

The **typedef** declares **VARRAY** to be the name of the type “variable length array of **n** elements of type **int**,” where **n** has the value it had at the point the **typedef** declaration was executed. **VARRAY** is used to declare **a1** and **a2** to be VLAs of **n** elements of type **int** where **n** has the value it had when the **typedef** was executed. Thus, if you make the call **ex2(5)**, **a1** and **a2** are both VLAs of five **int**s even though the value of **n** has been changed to 15 by the time **a1** and **a2** are declared. Of course, **a1** and **a2** can be used like any other arrays of five **int**s.

VLA **typedef**s follow the same rules as other VLA types. They can only appear in a block: they cannot appear at file scope. (VLA parameters are permitted because parameters are considered to be local to the function body.) The size of a VLA **typedef** is constant during its lifetime. The size is fixed when the **typedef** is executed. The size is no longer associated with the VLA **typedef** when the lifetime ends by either exiting the block or branching backwards in the block to a point before the **typedef** declaration [2]. VLA **typedef**s, like other VLAs, cannot be **struct** or **union** members.

Flexible Array Members

The last rule above about VLAs probably disappoints some of you. There are times when it would be useful for a VLA to be a **struct** member. While C99 does not permit that, it does permit a similar feature that standardizes an extension that some pre-C99 compilers permit in one form or another.

In C99, the last member of a **struct** may be an array with no bounds expression, called a flexible array member. A **struct** ending with a flexible array member allows you to have a **struct** object that ends with an array of any size you choose, if you are willing to do a little extra work. In fact, every different object with that **struct** type may end in a different-sized array.

The C99 compiler treats the flexible array member mostly like it is a zero-length array (ignoring the fact that zero-length arrays are invalid in C). So, the size of **struct** containing the flexible array member is identical to the offset in bytes of the flexible array member. If you just declare an object of a **struct** type with a flexible array member, you get an object that behaves normally except that no space is allocated for the elements of the flexible array member, and thus it is invalid to attempt to use those array elements.

If that was the full story of flexible array members, they would not be very useful. But, that brings us to that matter of extra work: if you allocate a **struct** with a flexible array member yourself on the heap, you control how much memory the object uses. If you allocate extra memory, it can be used for the elements of the flexible array member. It is valid to access any flexible array elements for which you allocated space. For example, if you allocate enough extra space for a three-element array, you can access elements zero through two of the flexible array member.

Listing 1 shows the use of a flexible array member. Some programming languages store strings not as a zero-terminated sequence of bytes like C, but as a count followed by the number of bytes specified by the value of the count. PL/I uses such a representation for its "varying strings." Java uses a similar representation (including an extra descriptor member) for string literals in class files. In Listing 1, the **struct PLIstring** gives the layout of a PL/I string. The member **s** is the flexible array member whose elements hold the characters in the string. The function **toPLI** converts the C string that is its argument into a newly allocated PL/I string on the heap. Note that a call to **malloc** passes not just the size of **struct PLIstring**, which is the size of the **struct** without any array elements, but it adds the size of the array that is to appear at the end of this particular **PLIstring** object, which is the value **len**.

If you run the program in Listing 1 using the command:

```
listing1 this is a test
```

you get the output (the first line is system specific):

```
count=12, s="listing1.exe"
count=4, s="this"
count=2, s="is"
count=1, s="a"
count=4, s="test"
```

There are various rules that flexible array members must follow:

- Only the last member of a **struct** may be a flexible array member. This rule follows from the fact that any extra space you allocate dynamically for the **struct** appears at the end of the **struct**, so that is where the flexible array should be.

- There must be at least one named member before the flexible array member. This avoids problems with zero-sized **struct**s, since the flexible array itself takes up no storage.
- A **struct** with a flexible array member may be a member of a **union**, but it may not be a member of another **struct**. A **union** containing a nested flexible array member may be a member of another **union**, but it may not be a member of a **struct**.
- You may not have an array whose elements are **struct**s with a flexible array member. You may not have an array whose elements are **union**s who have a nested flexible array member.

Unlike VLAs, the C implementation keeps no run-time information about the size of a flexible array member. It is the programmer's responsibility to allocate the space for the array and remember the number of elements in the array. If you assign a **struct** with a flexible array member or pass it as an argument to a function (not through a pointer), then the compiler generates code based on its compile-time information about the **struct** type. Since the compiler believes that the flexible array member has no elements, no elements will be copied during assignment. If you want to assign **struct**s that contain flexible array elements, you must make sure the target has the proper amount of memory allocated and then use **memcpy** or a loop to copy the flexible array elements.

As mentioned before, some pre-C99 compilers permit flexible array members. Some of those compilers use a slightly different syntax: rather than the flexible array member having no bounds inside the **[]**, the compilers permit the array bounds to be zero. (Officially, arrays of zero elements are not permitted in C.) Programs that use the **[0]** form of the extension can be converted to C99 merely by removing the **0**.

Unfortunately, in some cases, programmers relied on tricks before C99 to get the effect of flexible array members. Perhaps the most common form of that trick is to declare the fake flexible array with bounds **1**. When allocating the **struct** with **malloc**, extra space for an array of one less than the desired number of elements was allocated since the **struct** already had one element built in. While this technique is likely to work for most C and C++ implementations, it does break the rules. A small number of C implementations generate code to check if array indexes are in bounds, and they will complain about any index other than **0** being used with the fake flexible array. (Such checking is automatically turned off when using a real C99 flexible array.)

References

[1] Randy Meyers. "The New C: Why Variable Length Arrays?" *C/C++ Users Journal*, October 2001.

[2] Randy Meyers. "The New C: Variable Length Arrays, Part 2," *C/C++ Users Journal*, December 2001.

[3] Randy Meyers. "The New C: Variable Length Arrays, Part 3: Pointers and Parameters," *C/C++ Users Journal*, January 2002.

Randy Meyers *is a consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*